Universidade do Minho

Departamento de Informática

Programação Ciber-física

Trabalho Prático 2

Modelação e Análise de Sistemas Ciber-físicos com monads

André Gonçalves PG46525
Francisco Peixoto PG47194

10 de julho de 2022

Tarefa 1

A implementação dos monads para `ListDur a` foi a seguinte:
`fmap` percorre a lista e aplica `f` aos valores.

```
instance Functor ListDur where
   fmap f = LD . (map (fmap f)) . remLD
```

`pure` coloca `x` na lista com o `pure` de `Duration`.
`<*>` percorre as listas de `l1` e `l2` e aplica `<*>` de `Duration` a cada dois elementos de cada lista.

```
instance Applicative ListDur where
   pure x = LD [pure x]
   l1 <*> l2 = LD $ do x <- remLD l1
                       y <- remLD l2
                       return $ x <*> y
```

`return` é igual a `pure`.
`>>=` aplica `k` a cada valor nos elementos da lista de `l` e depois mapeia os valores na lista do resultado com `(x >>=) . const`, esta função soma as durações de `x` com cada elemento e mantém o valor deste.

```
instance Monad ListDur where
   return = pure
   l >>= k = LD $ do x <- remLD l
                     g x where
                        g x = map ((x >>=) . const) (remLD (k $ getValue x))
```

No problema dos aventureiros definimos a função que retorna o tempo que cada aventureiro demora a ultrapassar a ponte assim:

```
getTimeAdv :: Adventurer -> Int
getTimeAdv P1  = 1
getTimeAdv P2  = 2
getTimeAdv P5  = 5
getTimeAdv P10 = 10
```

Também definimos as seguintes funções e variáveis para utilidade:

```
-- lanterna
lamp :: Objects
lamp = Right ()

-- mover a lanterna
moveLamp :: State -> State
moveLamp = changeState lamp

-- lista dos aventureiros como objetos
advo :: [Objects]
advo = map Left [P1, P2, P5, P10]

-- lista dos aventureiros como objetos que podem atravessar a ponte
-- (estão no mesmo lado da lanterna)
canCross :: State -> [Objects]
canCross s = filter ((== s lamp) . s) advo

-- igual a getTimeAdv só que para os objetos
getTimeOAdv :: Objects -> Int
getTimeOAdv (Left a) = getTimeAdv a
```

Existem dois casos possíveis, o caso em que uma pessoa atravessa a ponte e o caso em que duas pessoas atravessam a ponte em conjunto. De forma a recriar este comportamento criamos estas funções que modelam cada caso.

`moveOne` gera as possibilidades de um aventureiro atravessar a ponte, para tal os aventureiros que estão no mesmo lado que a lanterna são filtrados (`lo`), essa lista é mapeada e gera as durações que os aventureiros demoram a atravessar.

`moveTwo` gera as possibilidades em que dois aventureiros atravessam a ponte, para tal usamos a função `paris` para obter os pares de aventureiros que podem atravessar a ponte e depois obtemos a duração resultante de cada par atravessar a ponte.

```
allValidPlays :: State -> ListDur State
allValidPlays s = manyChoice [ moveOne s, moveTwo s ]

moveOne :: State -> ListDur State
moveOne s = LD (map f lo)
               where
                  -- aventureiros do mesmo lado que a lanterna
                  -- lo :: [Objects]
                  lo  = canCross s
                  -- estado após aventureiro e lanterna atravessarem a ponte
                  -- f :: State -> Duration State
                  f x = wait (getTimeOAdv x) (return (changeState x (moveLamp s)))

-- retorna os pares possiveis de aventurerios dados
pairs :: [Objects] -> [(Objects, Objects)]
pairs [] = []
pairs (x:xs) = (map (\i -> (x, i)) xs) ++ (pairs xs)

moveTwo :: State -> ListDur State
moveTwo s = LD (map f po)
               where
                  -- pares dos aventureiros que podem atravessar a ponte
                  -- po :: [(Objects, Objects)]
                  po = pairs $ canCross s
                  -- durações dos pares após atravessarem
                  -- f :: (Objects, Objects) -> State
                  f (x, y) = wait (m x y) (c x y)
                  -- estado após dois aventureiros e lanterna atravessarem
                  -- c :: Objects -> Objects -> m State
                  c x y = return $ changeState x $ changeState y $ moveLamp s
                  -- tempo que demoram os dois aventueiros a atravessar
                  -- m :: Objects -> Objects -> Int
                  m x y = max (getTimeOAdv x) (getTimeOAdv y)
```

`exec` produz todas as possibilidades para `n` passos, recursivamente.

```
exec :: Int -> State -> ListDur State
exec 0 s = return s
exec n s = do s1 <- allValidPlays s
              exec (n-1) s1
```

`leq17` calcula se há alguma duração na lista gerada para os 5 passos em que o tempo é menor ou igual a 17 e todos os aventureiros estão na margem direita.

```
leq17 :: Bool
leq17 = any (\(Duration (x, s)) -> x<=17 && all s advo) (remLD (exec 5 gInit))
```

`l17` é similar a `leq17` , neste caso apenas testa se há algum elemento em que a duração é menor que 17 para os casos de 1 a 6 passos de execução.

```haskell
l17 :: Bool
l17 = any id [any f (remLD (exec i gInit)) | i <- [1..6]]
        where
           -- duração menor que 17 e todos atravessaram
           -- f :: Duration State -> Bool
           f (Duration (x, s)) = x<17 && all s advo
```

A partir do monad `ListDur` conseguimos provar que existe uma solução para **2.**, mas não para **3.**

Tarefa 2

Para a resolução e modelação do problema usamos o Haskell, e comparando com a modelação do problema no contexto das aulas usando o Uppaal determinamos as seguintes vantagens e desvantagens.

Uppaal: Vantagens

Na modelação em Uppaal conseguimos verificar e validar o modelo, bem com obter uma análise a fundo dos possíveis pontos de falha do mesmo, isto porque ao correr as simulações o Uppaal analisa aleatoriamente todos os casos possíveis e verifica e assinala eventuais deadlocks no programa.

As simulações também são um ponto positivo do Uppaal comparativamente ao Haskell visto que a sua execução é visual e ajuda numa melhor compreensão do verdadeiro comportamento modelado, passo a passo, o que resulta numa melhor e mais rápida correção de eventuais comportamentos inesperados.

A modelação em Uppaal também não implica diretamente a programação de algoritmos, visto que conseguimos simular problemas complexos apenas com as ferramentas simples que o Uppaal fornece, e tirar proveito dessas ferramentas com o uso de formas lógicas para verificar eventuais requisitos do sistema. Podemos verificar as propriedades temporais de um modelo facilmente a partir de fórmulas CTL.

Haskell: Vantagens

O Haskell necessita que o programador explicite corretamente os comportamentos do algoritmo face às diversas circunstâncias, o que leva ao programador ter uma perspetiva mais a fundo do problema, visto que a programação resulta do estudo da estratégia modelada.

O não-determinismo dos monads permite obter todas as possibilidades.

Haskell: Desvantagens

Por vezes a deteção dos pontos de falha advém de tentativa erro, o que não é de todo a melhor abordagem, e pode ser resolvida com a verificação do modelo em Uppaal.

O Haskell implica também a programação de todas as iterações necessárias para obter resultados, visto que estes são apenas os casos pré-planeados pelo programador, isto pode resultar numa falta de casos de teste suficientes para validar o programa.

Com Monads obtemos todos os estados possíveis, portanto podem existir estados repetidos e iterar sobre eles leva a computações desnecessárias.